UIUCDCS-R-75-745

AN INTERACTIVE TABLE-DRIVEN PARSER SYSTEM

by

Michael Harry Tindall

August, 1975

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-75-745

AN INTERACTIVE TABLE-DRIVE PARSER SYSTEM

by

Michael Harry Tindall

August, 1975

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was supported in part by the IBM Corporation, Thomas J. Watson Research Laboratory and the National Science Foundation, Grant No. EC-41511 and was submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science, August 1975.

ACKNOWLEDGEMENT

I would like to express my graditude to my thesis advisor, Professor Thomas Wilcox. He provided many suggestions and much useful guidance during my work on this project.

Thanks are also in order for Mike Milner and Bob Barnett who worked on the initial implementations of Fortran IV and COBOL. They provided very useful feedback on the operations and design of the overall system, and their assistance is greatly appreciated.

Also, a word of thanks goes to Cinda Robbins for her excellent typing of this manuscript.

TABLE OF CONTENTS

CHAPTER 1.

1. INTRODUCTION

1.1 Organization of this Thesis

This paper discusses the design and implementation of a table-driven syntactic parser with concurrent static semantic checks to be used in an interactive compiling environment. After a brief introduction in this chapter to the compiling environment in which the parser is to be used, and the the general operation of the parser system, Chapter 2 will present a model of the selected transition diagram parsing technique. Chapter 3 contains documentation on the parser's assembler language syntax source instruction specification. Chapter 4 contains detailed documentation on the form of each instruction that is used in the actual parser table. Chapter 5 describes the table maintenance system that is used by the compiler system, and the thesis concludes with a few comments about further refinements that can be made to this parsing system.

1.2 The Compiling Environment

Recently a project at the University of Illinois at Urbana-Champaign has been under way to automate the teaching of the basic Computer Science courses by utilizing the PLATO IV computer-aided instructional system that is being developed on this campus [4]. This computer system features an excellent graphical display terminal and fairly sophisticated computer-aided instructional software support for the writing of instructional lessons and the corresponding course curriculum [5]. The curriculum that is being implemented will teach new

computer science programming language concepts and constructs. Specific programming detail on a variety of languages (e.g., FORTRAN IV, PL/1, COBOL, BASIC) is available; students will progress at their own speeds through a fairly flexible course structure [2]. An important part of the system is an online compiler in which a student can easily and conveniently try out new programming constructs immediately after learning about them in an instructional lesson.

The remainder of this paper will discuss this compiler and, in particular, the parser system that is used in the compiler.

1.3 The Interative Compiler

A number of design criterion emerge from examining the environment for this compiler system. First, the compiler should be as interactive as possible to utilize the PLATO IV system effectively and to maintain a desirable computer-aided instructional environment for the student. To accomplish this, the compiler compiles character-by-character, that is, each single key press by the student using the compiler is examined immediately as the student types it in; thus the compiler keeps up completely with the student and detects programming syntax errors as soon as possible. The student is able to edit his program by moving a cursor through the program on the screen. The compiler moves with the cursor, compiling when the cursor moves forward in the program, and backing-up ("uncompiling", i.e., resetting the lexical and syntactical analyzers to previous states) when the cursor moves backward in the program. Thus, the compiler is highly interactive and easy to use.

A second design criterion for the compiler is that it be multilingual. To accomplish this, the compiler is completely table-driven; to allow

another language to be recognized by the compiler system and used by students, a language designer must merely fill in a new set of tables and provide an execution supervisor system for the actual interpretive execution of compiled programs. This paper is concerned with one of these compiler tables, namely, the syntax parser table.

A third design criterion for the compiler is that it provide a a high and sophisticated level of error diagnostics for the student when a syntactic or semantic error is detected in the program by the parser system. Since the intended users of the compiler are beginning students, the error messages must be direct and to-the-point. To accomplish this goal, an automatic, interactive error diagnostic system has been designed and implemented [6]. The important point for this discussion is that this automatic error system is driven by the compiler's syntax tables, that is, it is essentially language-independent. Thus, it is apparent that these syntax tables are a very important part of this compiler system.

We now examine a model for the transition diagram parsing technique used by the compiler system.

CHAPTER 2.

2. THE TRANSITION DIAGRAM PARSING MODEL

2.1 The Basic Model

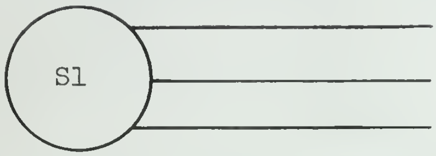
The syntactic analyzer used in this interactive compiler is based on the transition diagram systems first introduced by Conway [1], and recently formalized by Lomet [3]. A transition diagram system consists of a set of nested push-down automata (NPDA) that have the capability of invoking one another. The remainder of this section will present first an intuitive, graphical description of transition diagram systems, followed by a slightly more formal description of the transition diagram model.

A key concept of a transition diagram parser is that of the parser "STATE": the STATE is a descriptor that maintains information about what input has already been accepted and what further inputs would be acceptable to the parser. While "STATE" is a very important concept, it is a very easy thing to visualize and implement in a transition diagram system. For the rest of this paper, STATE refers to this "state of the parse".

The action of a transition diagram parser is to examine the "possible" or "acceptable" parsing options (determined by the current STATE and the (transition idagrams) along with the current input token; based on this information, the parser will <u>accept</u> the token by updating the STATE information and asking for a new input token, or <u>reject</u> the current input token and signal a syntactic/semantic error if the token does not satisfy any of the available options. Note that all the parsing options are defined

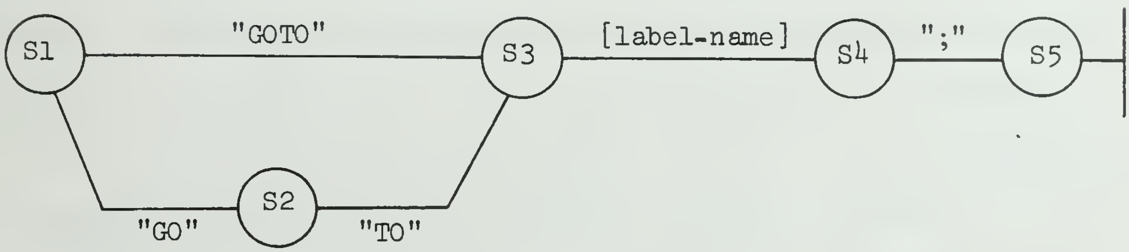
in terms of "tokens" only, that is, the tokens that are accumulated and output from the lexical analyzer. This process can be conveniently shown graphically as follows:

Let



denote STATE "S1"; each branch out of a STATE corresponds to a possible syntax option for that STATE: these branches are labeled with their particular syntactic option requirements (these labels will be described as the paper progresses).

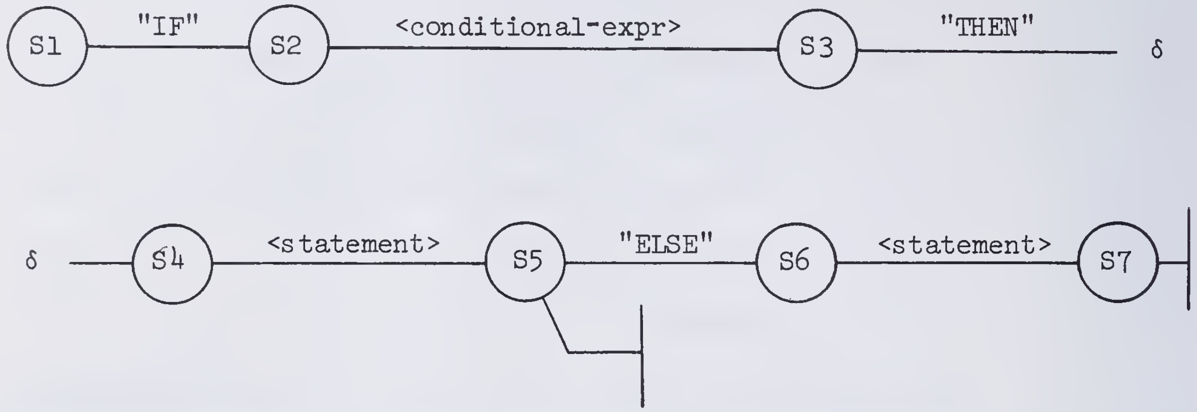
Then, a PL/I "GOTO" statement can be shown in transition diagram form as:



This is interpreted as: if the parser is in STATE S1 and the current input token is "GOTO", make the state transition to STATE S3; if the following token is a [label-name], move to STATE S4; if the token after

that is ";", accept it, and (in this case) accept an entire "GOTO" statement. Note that if none of the branch options for the current STATE satisfies the current input token, then that token is in error, and the normal parsing error condition should be signaled.

Another example is a PL/I "IF" statement:



In this case, notice the references to <conditional-expr> and <statement> as labels on option branches: this indicates that if, for example, the parser is in STATE S2, then when trying to accept the input token, it should refer to another transition diagram in the system, corresponding to <conditional-expr>; after a <conditional-expr> has been parsed and accepted, the parser should then return to STATE S3, having successfully satisfied the <condtional-expr> option branch out of STATE S2. This is an example of one transition diagram "invoking" ("calling", "referring to") another.

One more example is needed to illustrate another important feature of transition diagrams: a transition diagram system which has been invoked by another transition diagram has the capability of returning to one of a number of possible STATEs in the invoking transition diagram. A good example of where this is useful arises in trying to parse a (simplified)

PL/I <conditional-expr> as follows:

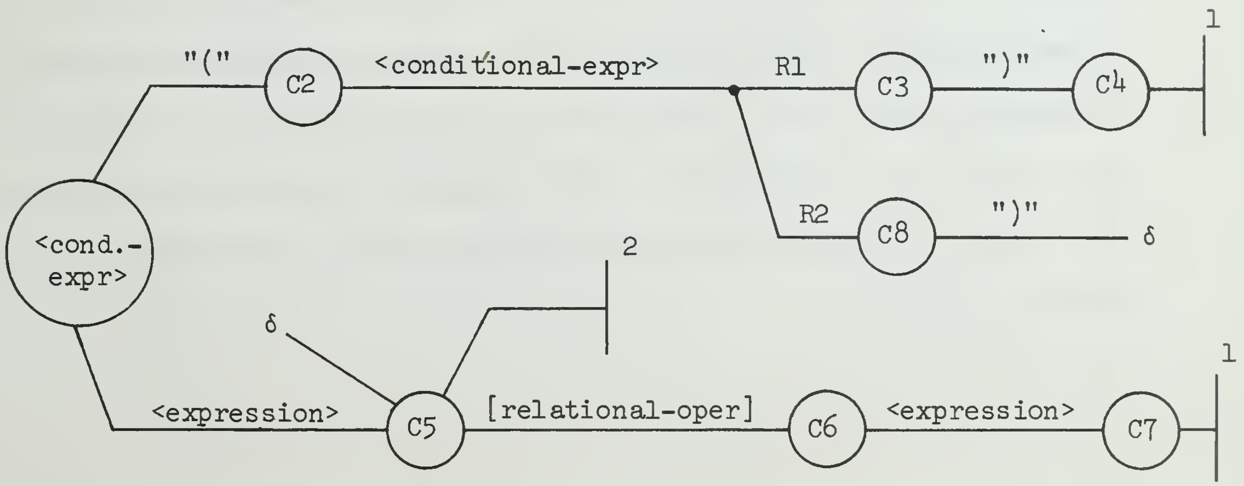
expression parentheses

(((I + 100) * J) = K)

conditional expression parentheses

When the initial parentheses "(" are examined, it is not known whether they are part of the overall conditional expression or are part of the inside simple expression. The technique to solve this ambiguity is to allow an "invoked" transition diagram to return to more than one STATE in the "calling" transition diagram, depending on what tokens are found later on.

The diagram for <conditional-expr> can be drawn as:



In this case, each time <conditional-expr> is invoked, 2 return STATEs are specified (i.e., R1 and R2); when the parser reaches $\dashv^{n}$, it returns to return-state n. Note that the way in which this <conditional-expr> has been drawn corresponds to assuming that the initial "(" belongs to the overall conditional expression; if it turns out they actually belonged to the first simple expression, then the parse is resumed at point δ, which accepts the assumed "conditional" parenthesis and continues parsing the simple expression.

As a more formal description, a transition diagram system consists of a set of nested push-down automata (NPDA) that have the capability of invoking one another. Each NPDA is capable of reading a portion of the input string and accepting or rejecting it. Lomet calls the NPDA that are capable of being invoked "submachines"; the initial STATE of a submachine is known as its "entry" STATE and a submachine is invoked by the use of its entry STATE number by another NPDA. This results in the invoking STATE being saved at the top of a parser stack and the parse resumed at the new entry STATE. Each submachine also contains one or more "exit" STATEs; when an exit STATE is reached, the top of the stack together with the particular exit STATE determine the STATE in the original invoking NPDA with which to continue the parse. An error in the parse is detected if an NPDA reads a token from the input string for which there is no corresponding STATE transition that the NPDA can make. The reader is referred to the discussion by Lomet [3] for further technical details.

2.2 Extensions to the Model

The preceding discussion in this chapter describes a parser model that is sufficiently powerful to recognize all deterministic context-free languages [3]. A few extensions to the model have actually been included in the compiler's parser system to enable the handling of context-sensitive static semantic requirements such as proper and consistent declaration of attributes for an identifier, consistent references to declared array variables, etc. These extensions include allowing auxiliary memory variables to be utilized by the parser (for operations such as counting the number of subscripts in an array reference), allowing the labels on any branch in a transition diagram to refer to any of these auxiliary variables or any symbol table field, and allowing special, language-dependent semantic-subroutines to be invoked at appropriate times during the operations of the parser system.

To summarize the activities of a transition diagram parser: it must be capable of requesting that a new input token be read in; it has to be able to test the current input token to decide which labeled branch to follow out of the current STATE; it must have facilities to manipulate a return-state parser stack; and, finally, it must be able to perform any semantic analysis that is required by a particular language.

CHAPTER 3.

3. ASSEMBLER SYNTAX SOURCE INSTRUCTION SPECIFICATION

3.1 Introduction and Chapter Organization

3.1.1 The Syntax Parser Table Description

A table-driven parser system has been designed that incorporates the actions described in the preceding chapter. The syntax table is actually an encoding of the programming language syntax productions in a small, interpretable instruction set. The parser of the compiler consists of a routine that interprets this instruction set in an appropriate way (this routine can be called the "table interpreter" or "table driven" routine).

The parser has control over and maintains certain tables and data structures within the compiler. All of these tables and structures are located in a region of memory that is referred to as the parser storage area. One particular variable that is maintained is the parser STATE variable. This variable always points to some instruction in the syntax table. As the input string is parsed, this state pointer variable is updated (i.e., moved to point to a different sequence) to reflect the current state of the parse.

The parser maintains two stacks. The first is the regular parsing return-state stack which is used when one NPDA invokes another. The other stack is called the "variable stack"; it is used by a language implementor to save miscellaneous information as the parse of an input string progresses; the language implementor has control over the

manipulation of entries in this stack: when to put new entries on the stack and delete entries from the stack (synchronized with PROC entry and exit), or change the value stored in an entry on the stack.

There are also certain tables that are maintained by the parser. These include the compiler's symbol-table and the compiler's block structure tables. Entries from these tables can be examined or modified by the parser with the complete control of the language implementor.

The action of the parser system in the compiler is to examine and interpret, beginning at some location in the syntax table (determined by the STATE variable), the table instructions that specify the acceptable syntax for the programming language. The instructions are interpreted sequentially unless a particular instruction modifies the parser's table instruction pointer.

For convenience in specifying the instructions in the parser table, an assembler language representation for each table instruction has been designed, and an assembler program is provided as part of the compiler system's maintanence utilities (chapter 5) that translates the assembler source language representation into the actual table instruction form that is used by the compiler system. The remainder of this chapter documents the assembler source language specification requirements, and chapter 4 documents the actual table form of the parser table instructions.

3.1.2 Chapter Organization

This chapter documents the assembler source language that is used to specify the syntactic requirements for a programming language.

Once a syntax source representation for the language has been prepared (using the compiler system's table builder option), an assembler program will translate the source representation into a compact table form to be used by the actual compiler.

The chapter is divided into three major sections as follows:

Section 3.2: Overall Organization of the Syntax Specification.

Discusses the proper ordering of instructions for the syntax specification; discusses the function and use of procedures (PROC-END blocks) in the specification; explains the form and the use of the mnemonic definition (DEFINE) instruction, the storage allocation (ALLOCATE) instruction, the different error NAME instructions, and the purpose and form of the FINAL PARSE STATE instruction.

Section 3.3: Parser Action Instructions.

Discusses the form and purpose of the instructions that are used to control the actions of the parser: SCAN, GOTO, CALLI, CALL, RETI, RET, BC, SEMA, and the auxiliary environment-changing instructions (ASSIGN, MASKON, MASKOFF, ADDIT, SUBIT).

Section 3.4: Description of Valid Parameters used in Action Instructions.

CLASS, PDN, UDN, PDSTP, UDSTP, defined constants, ALLOCATEd variables, Symbol-table entries, Block-table entries and TEMP variables.

3.1.3 Source Text Preparation Rules

The rules for preparing the syntax source specification for the assembler program are as follows:

1) Names:

All defined-names, variable-names, PROC-names and label-names discussed in this paper are of the form:

Any combination of 9 letters and number, with the first character being a letter (note that capital letters are acceptable, but they require 2 character positions in the name).

2) Form of instructions:

a) All instructions in section 3.2 are of the form:

<instruction> <u>eol</u>

that is, one instruction per line. The <u>eol</u> is inserted automatically by the editor program when a line is terminated. No explicit spacing is required within a line (free form, extra blanks ignored).

b) All instructions in section 3.3 (Action Instructions) are of the form:

<label> <instruction> <u>eol</u>

where the <label> is optional in all cases (except on instructions following a RETI, RET, GOTO, or unconditional branch (BC TRUE) instruction). The <label>, if present, marks the table-location for that instruction: control may then be passed to these <label>s via a BC, GOTO, or the multiple-return form of the CALLI and CALL instructions. It is suggested for readability (not required) that all <label>s begin at the left margin and all instructions begin at the normal tab position.

c) The following "brackets" are used in documenting the valid forms of instructions:

[...] : The specifications inside the [...] may appear 0 or 1 times.

{...} : The specifications inside the {...} may appear any number of times. (0,1,2, ...).

$\{...\}^n$: The specifications may appear at most "n" times (0,1,2, ..., n).

3) Throughout the remainder of this chapter all instruction keywords that are discussed will be CAPITALIZED for emphasis. However, as shown in the various examples given, these keywords are accepted in lower case only by the assembler program.

4) An asterisk appearing anywhere on a line causes the rest of the line to be treated as a comment. Comments may be used liberally.

3.2 Overall Organization of the Syntax Specification

3.2.1 Syntax Source Text Organization

The normal form for the syntax source text is to have the main procedure text first, followed by a sequence of PROC - END blocks.

The main procedure text contains all DEFINE instructions, followed by all global variable ALLOCATE instructions, followed by all error system instructions (CLASS NAME, MASK NAME, FIELDPDN, and ERROR MESSAGE), followed by the main parsing procedure (generally containing references to some PROCs).

The final PROC - END block is followed by END SYNA, which signals the physical end of the Syntax Source Representation.

SUMMARY:

{DEFINE instructions}

{Global ALLOCATE instructions}

{CLASS NAME instructions}

{MASK NAME instructions}

{ERROR MESSAGE instructions}

{FIELDPDN instructions}

.... main parsing procedure ....

{PROC - END blocks}

END SYNA

3.2.2 PROC - END blocks

The purpose of PROC - END blocks is to make transition diagram submachines well defined constructs.

FORM:

PROC <procname> [(<arg> {,<arg>}^{4})] [RETURN <#rets>]
[NAME (<error print name>)] eol

{local variable ALLOCATE instructions}

........proc instructions, including at least 1 RET instruction........

END PROC eol

ACTION:

<procname> is the procedure name.

The argument list is optional---however, a PROC must be consistently specified. The maximum number of arguments for a procedure is 5. The <arg>s (if present) are considered to be local variables to the

procedure (call by value, no result returned). They do not need to be ALLOCATEd (the assembler will automatically allocate space for any procedure <arg>s), however they may be included in an ALLOCATE statement inside the PROC if desired (the assembler accepts either implicit or explicit allocation in this case) (see section 3.2.6 for more information about local variable ALLOCATE instructions).

The MULTIPLE RETURN option is available to allow the parser to return to more than one state in the Syntax Specification after the procedure has been executed (see section 3.3.3- 3.3.6 for more information on CALLing and RETurning from procedures). The MULTIPLE RETURN option must be specified only if the PROC uses multiple returns. If the PROC uses multiple returns, the <#rets> parameter (which must be a numeric constant) specifies the number of locations the procedure may return to (the maximum number is 31).

The NAME option is for use by the compiler's autmatic syntax error system. The <error print name> chosen for a PROC should be a short logical name that describes the function of the PROC (examples: "statement", "expression", "declaration list", "array bound", etc.) (see section 3.2.7 for more information about <error print names>).

<u>EXAMPLE</u>:

```
proc    expr (type) return 2 name (expression)    eol
.
.
.
end   proc     eol
```

3.2.3 ENTRY Instruction

FORM:

ENTRY <Procname> [NAME(<error print name>)] eol

ACTION:

Defines an alternate entry point into the containing PROC. The <procname> is treated as a regular procedure name in CALL instructions. However, all of the attributes of the ENTRY <procname> are the same as those of the outer PROC:

--- number and order of parameter arguments;

--- number of multiple return points;

--- number of local variables.

Furthermore, none of these attributes can be specified at the ENTRY instruction ---- only at the containing PROC. The only unique attribute of an ENTRY <procname> is the (possibly) unique <error print name>.

EXAMPLE:

entry operand name (operand) eol

3.2.4 END SYNA Instruction

FORM:

END SYNA eol

ACTION:

Signals the physical end of the syntax source text to the assembler.

3.2.5 DEFINE Instruction

FORM:

```
DEFINE   <name> = <constant>  {, <name> = <constant>}   eol
```

ACTION:

This instruction is used to define mnemonic constants for the assembler to use. No table code is actually generated for this instruction.

Note that <constant> can be either a numeric constant or a previously defined mnemonic constant.

EXAMPLE:

```
define   ifx = 0, colonx = 104, dclvar = 10      eol
```

ALL DEFINE instructions must preceed all other statements of the syntax source specification.

ALTERNATE FORM:

Instead of <constant> above,

MASK (<12-bit mask of 0's and 1's>)

can be used. There must be exactly 12 bits specified between the parentheses.

This is a convenient way to define a particular mask bit pattern.

EXAMPLE:

```
define   numeric = mask (000001011111)      eol
```

It is also legal to combine the two forms of the DEFINE instruction on the same line.

EXAMPLE:

define ifx = 0, numeric = mask (000001011111), dclvar = 10 eol

3.2.6 ALLOCATE Instructions

FORM:

ALLOCATE <variable> {, <variable>} eol

ACTION:

Causes variable storage to be allocated on a stack in the compiler. These storage locations are referenced by using the name <variable>.

Note that the allocated storage will be a GLOBAL allocation if the ALLOCATE instruction comes at the beginning of the syntax program (i.e., before the first PROC - END block definition) and a LOCAL allocation (implying possibly recursive allocation) if the ALLOCATE instruction is within a PROC - END construct.

Global variables can be referenced from anywhere within the syntax specification, whereas local variables can be referenced only from within the PROC - END block in which they were allocated.

3.2.7 Error System Instructions

FORMS:

CLASS NAME <class-number> (<error print name>) eol

MASK NAME <mask-pattern>(<error print name>) eol

ERROR MESSAGE <error-number> (<overide error message text>) eol

FIELDPDN <pdn-number> {,<pdn-number>} eol

ACTION:

The purpose of the CLASS NAME and MASK NAME instructions, as well as the procedure NAME instruction (see documentation on PROC - END blocks) is to provide the compiler's automatic error analysis system with text to refer to CLASSes, MASKs and specific errors that appear in the parser environment.

The automatic error analysis system interacts with the user by suggesting different modifications that can be made to correct an error in the program. Many of these suggestions need to be made in the terminology of the programming language involved; these NAME instructions provide the correlation between things in the parser environment and the language terminology that describes these things.

Typical messages using these NAMES are:

"Replace $\boxed{\quad}$ with a relational operator."	Class Name
"Insert an array bound in front of $\boxed{\quad}$."	Proc Name
"Replace $\boxed{\quad}$ with a declared variable (numeric)."	Class, Mask Names

Each Mask pattern that is used in the mask form of the conditional branch instruction (see section 3.3.7.3) and each CLASS number should be given an associated error name.

The ERROR MESSAGE instruction is used to override the operation and analysis of the automatic error system. If the error number signalled by the parser (via a conditional branch (bc) instruction) matches the <error-number> given in an ERROR MESSAGE instruction, then the overide text is displayed to the user and all subsequent error analysis processing is aborted.

Finally, the FIELDPDN instruction is used to inform the error system about which pdn numbers (section 3.4.3.3) represent "field tokens"; each field pdn number should be included in a FIELDPDN instruction.

<u>EXAMPLES</u>:

class name punct (punctuation) <u>eol</u>

mask name numeric (numeric variable) <u>eol</u>

fieldpdn label, stmt <u>eol</u>

3.2.8 <u>FINAL PARSE STATE Instruction</u>

<u>FORM</u>:

FINAL PARSE STATE <u>eol</u>

<u>ACTION</u>:

Before the "execution" of a compiled user program can be attempted, there must be some way for the compiler supervisor system to verify that the program is indeed "complete" (since in the interactive compiling environment a user could request execution of an unfinished program).

This instruction indicates to the parser that it is in the program-accepting state; that is, the program can be executed if and only if the parser is in this state.

There must be exactly one accepting state specified in the Syntax Specification (i.e., the FINAL PARSE STATE instruction must occur exactly once).

3.3 Parser Action Instruction Description

This section describes the form and use of the assembler instructions that allow a language designer/implementor to fully specify the syntactic and semantic requirements of the language being implemented. These instructions constitute an implementation of the augmented parser transition diagram model discussed in Chapter 2 of this paper. The instructions allow invoking and returning from "submachines" (PROCs in this assembler language), examining the current input token (from the lexical analyzer) for validity, both syntactically and semantically (i.e., context-sensitive requirements), accepting the current token and requesting that a new token be input from the lexical analyzer, and finally, a few instructions allow changes to be made to the parser environment (i.e., symbol-table modifications or parser ALLOCATEd variable modifications).

One further note: many of the instructions to be described refer to general "parameters", which are the operand (s) used by the instructions. These parameters will be referred to as <parm>, or <parm1> and <parm2> in the form of the instructions. In all cases, these <parm>s will resolve to a memory location in the parser environment (like a symbol-table reference). These <parm>s are discussed in detail in section 3.4 of this paper.

The remainder of this section will discuss the Action Instructions: SCAN, GOTO, CALLI, CALL, RETI, RET, BC, SEMA, and the auxiliary environment-changing instructions (ASSIGN, MASKON, MASKOFF, ADDIT, SUBIT).

3.3.1 SCAN Instruction

FORM:

SCAN eol

ACTION:

Causes a return to LEXI for another token. When the next token comes in, the parser resumes parsing at the State following the SCAN instruction (i.e., at the table-location following the SCAN instruction's table-location).

3.3.2 GOTO Instruction

FORM:

GOTO <label> eol

ACTION:

Causes a SCAN instruction to be executed, followed by an unconditional branch to the instruction corresponding to <label> in the table.

EXAMPLE:

goto stmntl eol

3.3.3 CALLI Instruction

FORM:

CALLI <procname> [(<parm> {, <parm>})] [, THEN<label> {,<label>}] eol

ACTION:

This instruction is used to invoke a PROC with the name <procname>. (see section 3.2.2 and 3.2.3.)

Causes a return address (table-location) to be saved on the parser stack, passes the argument values to the proper local variables in <procname>, and resumes parsing at <procname> table-location.

When a return instruction (RETI, RET) is executed, the proper return label location is selected, the parser stack is popped, and parsing is resumed at the new location. If the PROC multiple-return option is not used, parsing is resumed at the location of the instruction following the CALLI (or CALL) instruction.

Note that the parameter argument values are passed into the PROC only, and that the final values are not passed back to the parameter argument upon returning from the PROC (i.e., call - by - value only).

Note also that all instances of the multiple return option for a PROC must be consistently specified (both in the CALLI (CALL) instructions and in the PROC - END definition).

<u>EXAMPLE</u>:

calli var (m,n), then lab1, lab2 <u>eol</u>

3.3.4 <u>CALL Instruction</u>

<u>FORM</u>:

Same as CALLI instruction

<u>ACTION</u>:

Causes a SCAN instruction to be executed, followed by a CALLI instruction.

<u>EXAMPLE</u>:

call subscr <u>eol</u>

3.3.5 RETI Instruction

FORM:

RETI [<return-number>] eol

ACTION:

Causes a return from a previously invoked PROC (see section 3.3.3).

If the PROC has any locally-ALLOCATEd variables, the space is deallocated from the variable-storage stack in the parser environment. Then the return address (table location) is popped off of the regular parser stack.

If the multiple-return option is not used by this PROC, parsing resumes at the popped return address location.

If the multiple-return option is used by the PROC, the "return-number"th <label> given in the original CALLI (CALL) instruction is selected and parsing resumes at the table-location corresponding to this selected <label>.

Note that <return-number> must be a constant (or a mnemonic defined constant) in the assembler.

EXAMPLE:

```
reti 2 eol
```

3.3.6 RET Instruction

FORM:

Same as RETI Instruction

ACTION:

Causes a SCAN instruction to be executed, followed by a RETI instruction.

EXAMPLE:

ret eol

3.3.7 BC Instruction

3.3.7.1 Normal BC Instruction

FORM:

BC <relation-type>, <parm1>, <parm2>, <true-option> eol

where

<relation-type> ::=EQ | NE | GT | GE | LT | LE

<true-option> ::=<label> | ERROR [<parm>]

ACTION:

Causes the 2 <parm>s to be compared according to <relation-type>.

If the comparison is false, the <true-option> is ignored and control falls through to the next instruction in the table.

If the comparison is true, the <true-option> is taken:

If <true-option> is a <label>, then parsing resumes immediately at the table-location corresponding to that label.

Otherwise, <true-option> is a syntax error indicator; this causes the parser to halt its operation and compiler control is passed to the compiler's error analysis system, with an Error Number equal to the value of the <parm> (if specified).

In the compiler's automatic error analyzer, the Error Numbers are ignored unless an ERROR MESSAGE instruction for the particular Error Number has been included with the Syntax Specification (see section 3.2.7 of this paper).

In a hand-coded compiler error system, the Error Number can be used to display a unique error message to the user. The BC instruction is the only instruction that is available to signal that a syntactic/semantic error has occurred.

Note that <parml> cannot be a constant or mnemonic defined constant.

EXAMPLES:

```
bc    ne, pdn, colonx, notlab          eol
bc    eq, class, dclvar, assign        eol
bc    ne, pdn, thenx, error 10         eol
```

3.3.7.2 Unconditional Branch Instruction

FORM:

```
BC    TRUE,  <true-option>       eol
```

ACTION:

Causes the <true-option> to be executed exactly as if the instruction was a normal BC instruction whose parameter comparison was TRUE.

EXAMPLE:

```
bc true, looplab                 eol
```

3.3.7.3 Attribute-Checking BC Instruction

FORM:

Same as the normal BC instruction, except that

<relation-type> ::=MASK, <mask-type>

<mask-type> ::=NOTANY | NOTALL | ANY | ALL

ACTION:

Most context-sensitive language requirements can be viewed as "attributes" of the particular tokens (both pre-defined and user-defined tokens) used in a user's program. In this compiler system, each symbol-table field contains 12 bits; although some of the symbol-table fields have very specific builtin uses (i.e., PDN for pre-defined symbols, and CLASS for both pre-defined and user-defined tokens), some of the remaining fields have no builtin use (for example, the UDN field for user-defined tokens). It is suggested that the language designer select an unused symbol-table field (such as UDN) and let each of the 12 bits in the field represent a different attribute that a user-defined token may have. Specific attribute bits may be turned on or off using the MASKON and MASKOFF instructions (see section 3.3.9). The existence of an attribute for a token can then be checked using the MASK form of the BC instruction.

The 2 parameters are compared according to the specified <mask-type>. For example, if <mask-type> is NOTANY, then the comparison is TRUE if NOTANY of the bits that are 1 in <parm2> are also 1 in <parm1>, and FALSE otherwise.

Note that it is possible to check for 1 attribute bit being on or off, or any combination of attribute bits being on or off. This allows for example, the grouping of two attributes like "FIXED and "FLOAT" together for certain types of tests (like the attempted declaration of an attribute "CHARACTER"); it may not be important which of the grouped attributes conflicts, but simply that a conflict exists.

The compiler's automatic error analysis system views the MASK form of the BC instruction as specifying an attribute check of the current token being examined by the parser. It is possible to give each particular bit-mask that is used as a <parm2> in a BC MASK instruction a unique error print name (see section 3.2.7); this print name will then be used in any generated diagnostic messages that involve the bit mask.

If a hand-coded error system is used, appropriate unique Error Numbers must be used as in a normal BC instruction.

EXAMPLES:

Assume that at some point in the syntax specification, the only valid syntax option is a numeric declared variable. Then if

dclvar :: = a constant whose value is the CLASS for a user declared variable, and

num :: = a constant whose value is the bit-mask for the numeric attribute(s),

the following instructions perform the required check:

```
        .
        .
        .
        bc ne, class, dclvar, error 1 *dclvar required here...
        bc mask, notany, udn, num, error 2 *numeric-type required...
        .
        .
        .
```

As another example, assume that a new user identifier is being declared. Let

```
        varattrib  :: = accumulated attribute bit-mask for new identifier, and
        conflict   :: = the conflicting attribute bit-mask for a new attribute
                        the user is trying to add for the identifier.
```

The following instruction check the validity of the new attribute:

```
        .
        .
        .
        bc mask, any, varattrib, conflict, error 3 *attributes conflict.
        .
        .
        .
```

3.3.8 SEMA Instruction

FORM:

SEMA <sema-number> [(<parm> {, <parm>})] eol

ACTION:

Occasionally certain unique, non-standard operations must be performed by the parser. The SEMA instruction allows a language implementor to write a regular TUTOR unit to perform these operations, and then have the parser execute these TUTOR units at appropriate times.

<sema-number> must be a constant or a mnemonic defined constant, between 1 and 15. The language implementor supplies TUTOR units named SM1 to Sm15 for the compiler. Upon execution of the SEMA instruction in the syntax table, the parser will do the correspondingly numbered TUTOR unit. After the TUTOR unit is finished, the parser resumes parsing with the next instruction in the table.

Up to 5 parameters may be passed to semantic routines (note that the assembler performs no check for inconsistent number of arguments for different uses of a particular semantic routine number).

To use the parameters in the TUTOR unit:

The parameters are passed as addresses into the parser storage environment through the use of 5 specially located variables in the parser storage: they are located at parser storage locations (ps_prm + i), where i is the parameter's number (1 - 5) and ps_prm is a compiler system defined constant.

Therefore, to use the address of the argument passed through parameter i, reference : ps (ps_prm + i). This address is some location in the parser storage environment (for example, a particular symbol-table field for the current token in the parser).

To use the value of the argument passed through parameter i, reference: parm(i) which is defined in the compiler as ps(ps(ps_prm + i)).

The only exception to the above is that constants (or mnemonic defined constants) passed as parameters are passed as just the value of the constant (i.e., reference this value as ps(ps_prm + i)). It is up to the language implementor to know and keep track of which parameters in a semantic routine are passed as constants and which are passed as addresses, and to specify the parameters consistently for each routine.

EXAMPLES:

```
sema  getdopev (udstp, st_dvl(udstp))          eol
sema  opendblk           eol
sema 4          eol
```

RESTRICTIONS ON THE USE OF SEMANTIC ROUTINES:

There are a few restrictions that must be placed on the use of semantic routines:

(1) Tracing of any changes made to parameters passed as addresses:
In order to allow the compiler to properly "backup" if the user edits the program being written (since this is an interactive compiling environment), anytime that the value of an address in the parser storage environment is changed, the old value must first be traced using the compiler unit TRACE, which has as its one argument the address to be changed. For example, if semantic routine parameter number 2 is passed as an address, and the semantic routine decides to change the value at that address, the following TUTOR code is needed:

```
do   trace (ps(ps_prm +2))   $$ trace old value
calc parm  (2) ← 'the new value'
...
```

(2) Handling of syntactic/semantic errors detected within a semantic routine:

If an error is detected in a semantic routine, it is not permitted for the semantic routine TUTOR unit itself to execute a transfer of control from the parser to the error analysis system. Instead, all exits to the error system must come through having the parser execute a BC instruction that has a <true-option> of the ERROR form (see section 3.3.7 of this paper).

The easiest way to do this for an error that is detected within a TUTOR semantic unit (note that this type of error checking in a TUTOR semantic unit is very non-standard--nearly all detectable syntactic/semantic errors can be detected through the use of appropriate BC testing instructions) is to use a temporary variable TEMPi (see section 3.4.3.5 of this paper) that is returned from the TUTOR unit as either 0 (everything is ok), or non-zero (error detected--the non-zero value can be an appropriate Error Number), and then the TEMP variable can be checked in the instruction following the SEMA instruction:

```
...
sema    some-number (some-parameters)         eol
bc  ne, templ, 0, error templ     *gives proper error number
                                  *for hand-coded error system.
...
```

3) Variables that may be referenced within a TUTOR semantic routine: Any parser storage address that must be referenced within a TUTOR semantic routine unit must be passed through the parameter list. Although this is the only way to reference an ALLOCATED variable in the Syntax Specification, this restriction also includes the special parser storage locations like PDN, CLASS, UDN, PDSTP and UDSTP, even though these locations are also defined directly within the compiler system itself (see section 3.4.3 of this paper for a description of these special locations).

This restriction is imposed by the compiler's automatic error analysis system.

4) Modifying a special parser storage location (section 3.4.3) in a semantic routine requires some care by the language implementor. Since the special locations are actually just duplicate, easily referencable copies of some of the fields in the symbol table entry for a token, any changes to either of the two corresponding locations should be accompanied by a change to the other location also. Note that only the symbol-table fields' values need to be traced, and not the special locations.

3.3.9 ASSIGN, MASKON, MASKOFF, ADDIT, SUBIT Instructions

FORMS:

ASSIGN <parm1>, <parm2> eol

MASKON <parm1>, <parm2> eol

MASKOFF <parm1>, <parm2> eol

ADDIT <parm1>, <parm2> eol

SUBIT <parm1>, <parm2> eol

ACTION:

Each of these instructions is used to change the value of the parser storage location for <parml> to <parm2>, or some function of <parml> and <parm2>.

The <parm>s are described in section 3.4 of this thesis.

ASSIGN:

Sets the value of <parml> to the value of <parm2>.

MASKON, MASKOFF:

Sets (on, off) all bits in <parml> corresponding to 1's in <parm2>; does not change bits in <parml> corresponding to 0's in <parm2>.

ADDIT, SUBIT:

(Adds, subtracts) the value of <parm2> (to, from) the value of <parml>.

Each of these instructions will trace the old value of <parml> before executing the instruction (except for the TEMP variable, and also PDN, CLASS, UDN, UDSTP, PDSTP variables, which do not need to be traced (see sections 3.4.3.1 - 3.4.3.4 for more details)).

IMPORTANT NOTE:

If <parml> is CLASS, UDN, PDN, then both the special parser storage location and the corresponding symbol-table field are modified (see section 3.4.3.2 - 3.4.3.4 for more details).

EXAMPLES:

```
assign class, dclvar      eol
addit numsubs, 1          eol
```

3.4 Description of Valid Parameters Used in Action Instructions

Most of the Parser Action Instructions have one or more <parm>s, that is they have "parameters" or "operands" associated with them (specifically, BC, CALLI, CALL, SEMA, ASSIGN, MASKON, MASKOFF, ADDIT SUBIT). This section documents the form and uses of the different <parm>s that are available in the parser environment.

3.4.1 Numeric Constants

Numeric (integer) constants or mnemonic DEFINEd constants are legal parameters (except as <parm1> of a BC (see section 3.3.7) or an auxiliary environment-changing (section 3.3.9) instruction). The constant's value is packed directly into the parser table. The only illegal constant value is 4095 (octal o7777), which is reserved for use by the compiler's automatic error analysis system.

3.4.2 ALLOCATED (Global and Local) Variables

GLOBAL: Global variables are legal parameters anyplace in the Syntax Specification. The parser storage direct address of the Global variable (known at assembly time) is packed into the parser table.

ALL GLOBAL variables must be ALLOCATEd at the beginning of the Syntax Specification (directly following any DEFINE instructions)----(see section 3.2.6).

LOCAL: Local variables are legal parameters any place within the PROC - END block in which they are ALLOCATEd. An

indexed parser storage address (relative to the parser's variable storage stack) is packed into the parser table. LOCAL variables must be ALLOCATED at the beginning of the particular PROC - END block in which they are active; note that a PROC's parameters are implicity ALLOCATEd Local variables (see section 3.2.6).

3.4.3 Pre-defined Parser Variables

There are a number of pre-defined parser storage variables that can be tested or otherwise used as legal parameters anywhere in the Syntax Specification. In all cases, the parser storage direct address (known prior to and during assembly time) is packed into the parser table.

3.4.3.1 PDSTP, UDSTP: Pre-defined and User-defined Symbol Table Pointers

In this compiler system, the symbol-table is logically divided into 2 parts: one part consists of all of the pre-defined tokens that belong to the language being implemented (i.e., the reserved(or unreserved) keywords, punctuation symbols, operators, etc.); the second part of the symbol table consists of any tokens that a user may have used in writing a program and that either do not have a corresponding pre-defined entry, or else are used in a context that is different from that of the corresponding pre-defined entry, (for example, a declared variable "IF" in PL/1).

The PDSTP and UDSTP variables always contain the values of the appropriate symbol-table pointers for the current token that the parser has received from the lexical analyzer. If the current token has a pre-defined and a user-defined symbol table entry, then PDSTP and UDSTP point to these entries, respectively. If the current token has only a pre-defined entry, then both PDSTP and UDSTP point to this pre-defined entry. If the current token has only a user-defined symbol entry, then UDSTP points to this entry and PDSTP is essentially null (it points to an empty pre-defined symbol-table entry that no token can ever resolve to).

Note that the only way for a token to have both a pre-defined and a user-defined symbol-table entry is for the language implementor to specifically provide an appropriate SEMAntic routine (see section 3.3.8) that actually creates the user-defined entry from the pre-defined entry; the compiler's symbol-table manager will not create a user-defined entry automatically for a token that resolves to a pre-defined location.

See section 3.4.4.1 for a description of how to reference particular symbol-table fields, given UDSTP or PDSTP.

3.4.3.2 CLASS: The syntactic/semantic class of the current token

The CLASS variable is defined as ST_TYP (UDSTP) (see section 3.4.4.1). For pre-defined tokens, CLASSes will usually be things like "statement keywords", "relational operators", "attribute keywords", etc. For user-defined tokens, CLASSes will be things like "declared variable", "labels", "undeclared variable", etc. These class values are so commonly referred to in parsing a language that the parser maintains a special

parser storage location that contains the CLASS of the current token that is being examined.

If the current token has only a pre-defined symbol-table entry, then CLASS is set to the ST_TYP field of this pre-defined entry. If the current token has a user-defined symbol-table entry (or both a pre-defined and a user-defined entry), then CLASS is set to the ST_TYP field of this user-defined entry.

Note that the CLASS of a user-defined token may be changed and updated through the use of any of the auxiliary instructions (see section 3.3.9). If the lexical analyzer accumulates a "new" token, that is, a token that has no symbol-table entry, then the symbol-table manager will automatically create a new user-defined symbol-table entry for the new token, and the CLASS (ST_TYP) field of the new entry will be set to the default CLASS value specified by the lexical analyzer.

Note that all CLASS values used in a particular language implementation should be given CLASS NAMEs for the compiler's automatic error analysis system (see section 3.2.7).

3.4.3.3 PDN: Pre-defined Number

PDN is a unique identification number for a pre-defined token. The PDN variable is defined as ST_IDN (PDSTP) (see section 3.4.4.1 for a description of symbol-table fields). Each token that is entered in the pre-defined symbol-table by a language implementor should be given an identification number that can be checked by the parser (using BC instruction, section 3.3.7) when a particular pre-defined token is a valid parse option for the current parser State and parser environment. The same

identification number may be given to more than one pre-defined token to allow for pre-defined synonyms or abbreviations (such as "DECLARE" and "DCL" in PL/I).

Note that PDN is always based on PDSTP, so if no pre-defined symbol-table entry exists for the current token being examined, the value of PDN is null, that is, it will match nothing.

3.4.3.4 UDN: User-defined Number

UDN is the value of ST_IDN (UDSTP), which does not have a pre-reserved meaning in the symbol-table (that is, it is available for use by a language implementor in whatever way is desired).

The UDN variable (i.e., the ST_IDN (UDSTP) symbol-table field) is best used as an attribute field for user-defined tokens. Each bit in the field (there are 12 of them) can be used to denote a particular attribute in the programming language that a user-defined token may assume. These attribute bits can be turned on or off using the MASKON and MASKOFF auxiliary instructions; also the attributes of a particular token can be tested for consistency using the BC MASK form of the BC instruction. See section 3.3.7 for a more complete discussion of the handling of attributes by the parser.

3.4.3.5 TEMP: Used as temporary computation variables only

There are 5 temporary variables available for use by the language implementor: TEMP1, TEMP2, TEMP3, TEMP4, TEMP5. These variables can be used to temporarily save any value in the parser environment. The most important restriction on the use of these TEMP variables, however,

is that they may NOT be used to save a value if the parser returns to the lexical analyzer for a new token: these variables are only valid between returns to the lexical analyzer by the parser.

The reason for this is that the value changes of these variables are not traced by the parser, so that program editing performed by the user will not properly restore the values to the variables. The variables are most useful for returning error indicators from SEMAantic routines.

3.4.3.6 BLOCK:

Contains the current symbol-table block number that is being used by the symbol-table and the parser.

3.4.3.7 ITPTP: Intermediate Text Parser Token Pointer

Contains the current location of the intermediate text pointer. This value is useful for saving information about locations in the intermediate text that will be needed when the user executes the program (like "label" locations, or Subprogram entry points, etc.).

3.4.4 Table References

There are two tables in the parser environment that may be accessed as general parameters: the symbol-table and the block-structure table. Assembler references to these tables all follow the same general form:

FIELD-NAME (<parmx>)

where

FIELD-NAME is the particular table field name, and

<parmx> is the table index pointer.

The following restrictions apply to <parmx>:

a) <parmx> may be ALLOCATEd variable (section 3.2.6);

b) <parmx> may be pre-defined Parser Variable (section 3.4.3);

c) <parmx> may be a table reference again, but then the parameter for the new table reference must be either a Global ALLOCATEd variable or a pre-defined Parser Variable only (i.e., at most 2 levels of indexing are allowed).

3.4.4.1 Symbol Table Fields

Each symbol-table entry in the symbol-table contains 10 different fields; in addition, a dope-vector symbol-table entry may be associated with a regular symbol-table entry through the use of the ST_DVL fields in the regular entry (see the discussion of the ST_DVL field below); these dope-vector entries contain 5 fields.

ST_TYP: This field is used to contain the syntactic type for the token. For user-defined tokens, that is, those tokens which are accumulated by the lexical analyzer and that initially have no symbol-table entry, the symbol-table manager in the compiler will assign a symbol-table entry and will initialize this ST_TYP field to a value supplied by the lexical analyzer.

This is so important to the operation of the parser that a special pre-defined parser storage variable has been provided to hold the ST_TYP of the current token being examined by the parser. This special field is the CLASS field; see section 3.4.3.2 of this paper for

more information about the CLASS pre-defined location.

ST_IDN: For pre-defined symbol-table entries, this field contains a unique identification number for the pre-defined token. This ST_IDN number is so important that a special pre-defined parser storage location has been provided to hold the ST_IDN number for the current token being examined by the parser. This special field is the PDN field; see 3.4.3.3 of this paper for more information about the PDN pre-defined location.

For user-defined symbol-table entries, the ST_IDN field does not have a pre-reserved meaning in the symbol-table; therefore, a language implementor may use the field in whatever way it desired. However, it is <u>strongly</u> suggested that this field be used as an attribute field for the user-defined tokens (section 3.3.7 of this paper discusses more completely the handling of attributes by the parser system). The special pre-defined parser storage variable UDN has been provided to hold the ST_IDN value for user-defined tokens (section 3.4.3.4).

ST_OFF:

ST_LEN: These 2 fields in the symbol-table have no pre-reserved meaning, and a language implementor may utilize them in whatever way is desired (for appropriate tokens, these fields should be used for the offset and length of storage at runtime).

ST_BLK: This field is unused for pre-defined symbol-table entries. For user-defined entries, this field contains the block number of the inner-most block where the corresponding token was first used; this number can be used to access the parser's Block Tables (section 3.4.4.2).

ST_SIB: This field is used to link together all user-defined symbol-table entries in the same Block.

ST_LNK: The field is used to link together <u>all</u> symbol-table entries, regardless of what Block they are in.

ST_PDE: This field points to the pre-defined symbol-table entry for a token; if no pre-defined entry exists, it points to a special "null" pre-defined symbol-table entry.

ST_NTP: This field points to the NAME table entry for the token. Note, that the NAME table itself is not considered part of the parser environment, and is thus not accessible directly as a parameter (however, a SEMAntic routine may reference the NAME table fields).

ST_DVL: This field is unused for pre-defined symbol-table entries, and also for user-defined entries that have no corresponding dope vector entry.

If a user-defined symbol-table entry requires a dope vector, the language implementor must provide a SEMAntic routine (section 3.3.8) that will get a dope vector entry from the symbol-table manager and then set ST_DVL to point to this dope vector entry. Then any dope vector field can be referenced indirectly through this ST_DVL field.

Dope Vectors: A Dope Vector contains these fields: ST_DIM, ST_LB1, ST_LB2, ST_UB1, ST_UB2. It is very important to remember that a Dope Vector is only available for a token if it has been explicitly requested (see ST_DVL above). Then the Dope Vector fields must be referenced indirectly through the regular symbol-table entry's ST_DVL field. Note that the following definitions of the Dope Vector fields

are actually only suggested; a language implementor may use the fields in any way that is desired.

ST_DIM: Contains the number of dimensions for an array. The number can be either 1 or 2.

ST_LB1:

ST_LB2: Contains the lower bounds of the dimensions of an array.

ST_UB1:

ST_UB2: Contains the upper bounds of the dimensions of an array.

3.4.4.2 Block Table Fields

The compiler's symbol-table and symbol-table manager were designed to allow normal block-structuring with respect to user-defined tokens to be possible. For this purpose, two block structure tables, referenced as

BT_LNK and BT_KID

are included as part of the parser environment. Both of these tables are indexed by an appropriate Block number, usually contained in the parser variable BLOCK (section 3.4.3.6).

The BT_KID table entry for a Block consists of a pointer to a symbol-table entry for a token that has been declared by the user in that block; then the rest of the symbol-table entries also in the same block are chained together through the ST_SIB field in the normal symbol-table entries. A zero ST_SIB field ends the chain.

The BT_LNK table entry for a Block contains the number of the block containing the current block. This information can be used to find the outer block declarations of a name. It is also useful for

maintaining correct nesting of variable definitions at runtime.

The language implementor has control over the opening and closing of blocks. To (open, close) a Block, it is necessary to provide compiler units (blkbgn, blkend) to be executed in TUTOR. These units will change the current value of BLOCK (section 3.4.3.6) and perform other appropriate modifications for the desired action.

EXAMPLES:

The following are all legal table references; the references to the dope vector fields assumes that a dope vector has been allocated in the symbol-table for the token (see discussion above on ST_DVL field).

```
st_typ (udstp)  ,corresponds to CLASS.
st_idn (pdstp)  ,corresponds to PDN.
st_idn (udstp)  ,corresponds to UDN.
st_off (udstp)
st_len (udstp)
st_blk (udstp)
st_dvl (udstp)
st_dim (st_dvl(udstp))
st_ub2 (st_dvl(udstp))
bt_kid (block)
bt_kid (outerblk)   , where outerblk is an ALLOCATEd variable
                      whose value is a valid block number.
st_typ (var)  ,where var is an ALLOCATEd variable whose value
                is the symbol-table pointer (udstp) for a token.
```

CHAPTER 4.

4. THE PARSER TABLE INSTRUCTION FORMS

4.1 Notation

This chapter will describe the actual instruction forms that appear in the syntax parser table. It is these instruction forms that are examined and interpreted by the parser in the compiler to determine whether a user's program is syntactically and semantically correct.

The basic instruction size, like the word size in the parser storage area, is 12 bits; this chapter will refer to these 12-bit packages as "words". Some instructions require exactly 1 word in the table; others require exactly N words, where N>1; and some instructions require a variable number of words, depending on the value of fields within the initial words of the particular instruction.

The following notation will be used on the following pages:

[state]: state number, points to an instruction word location somewhere in the syntax table (examples are [call-state], [return-state], [true-state]).

<parm>: an instruction parameter or operand. A <parm> is either a number (numeric constant) packed directly in the table, or else the address of a variable in the parser storage area. Thus a <parm> can take 1 word (for numeric constant packed in the syntax table, or a direct address of a number in the parser storage area), or 2 words (an indexed address of a number in parser storage), or 3 words (for a doubly-indexed address of a variable in the parser storage area).

The forms of the <parm>s used in particular instructions are determined by the actual instruction numbers (op codes).

4.2 The Table Instructions

4.2.1 SCAN Table Instruction

TABLE FORM:

/ 0 0 0 0 0 0 0 0 0 0 0 0 /

ACTION:

Sets the parser's STATE variable to point to the following instruction in the table, and returns to the lexical analyzer for another token. When another token has been input to the lexical analyzer, the symbol-table manager determines the symbol-table location(s) for the new token and parsing resumes with the saved STATE instruction.

4.2.2 ALLOCATE Table Instruction

TABLE FORM:

/ d d d d d d d 0 0 0 0 1 /

ACTION:

Pushes ddddddd new entries on the parser's variable stack. No initialization of the values of these entries is performed; the stack pointer is merely incremented.

4.2.3 DEALLOCATE Table Instruction

TABLE FORM:

/ d d d d d d d 0 0 0 1 0 /

ACTION:

Pops ddddddd entries off of the parser's variable stack. The value of each entry that is popped off is traced, and the stack pointer is decremented.

4.2.4 Call Table Instruction

TABLE FORM:

/ d d d d d d d 0 0 0 1 1 /	word 1	
/ x x x x x x x x x x x x /	word 2	
/ y y y y y y y y y y y y /	word 3	
.		
.	to	: Optional
.		
/ y y y y y y y y y y y y /	word 3 + ddddddd - 1	

ACTION:

xxxxxxxxxxxx : [call-state] table location.

ddddddd : number of multiple return points for the called procedure. If ddddddd = 0, then the procedure is normal (not multiple return).

yyyyyyyyyyyy : the multiple return table locations (if any).

Pushes the table location for word 3 onto the parser's return-state stack. Then resumes parsing immediately at the table location [call-state]. See section 4.2.5 for a discussion of the actions that occur when a procedure is returned from via a RET instruction.

Note that any arguments that are passed into the call-procedure are passed by using ASSIGN instructions immediately proceeding the CALL instruction (see section 4.2.8.3 for a discussion of the ASSIGN instruction).

4.2.5 RET Table Instruction

TABLE FORM:

/ d d d d d d d 0 0 1 0 0 /

ACTION:

Causes a return from the current procedure to where ever it was called from (see section 4.2.4 for a description of the CALL instruction).

Pops the return-address table location off of the parser's stack; this location corresponds to the work in the CALL instruction that follows the [call-state] number.

If ddddddd = 0, then the procedure does not use the multiple-return feature, so the popped address corresponds to the location of the instruction that follows the original CALL instruction. Therefore, parsing is resumed immediately at this location.

If ddddddd> 0, the procedure does use the multiple return feature, and ddddddd is the number of the multiple return address to use. For this case, the multiple return addresses are all packed directly in the parser table following the CALL instruction's [call-state] number; the return address table location that was popped off of the parser's stack points to the first of these multiple return addresses. Therefore, the ddddddd'th multiple return address is selected from the parser table, and parsing resumes immediately at this address.

Note that when the parser's stack is popped, the old entry at the top of the stack, as well as the stack pointer itself, are traced.

4.2.6 SEMA Table Instruction

TABLE FORM:

/ d d d d d d d 0 0 1 0 1 /

ACTION:

Causes the ddddddd'th TUTOR-coded semantic routine (supplied by the language implementor for each language) to be executed. The semantic routines are named Sm1, Sm2, ..., Sm15.

Any parameters for the semantic routine are passed in the "parm" parser storage variables (see section 3.3.8) by using the appropriate PASSIGN instruction (section 4.2.8.3) immediately preceding the SEMA instruction.

4.2.7 Unconditional Branch Table Instruction

TABLE FORM:

/ m m 0 0 0 0 0 0 0 1 1 0 /	word 1
/ [true-option] /	word 2 (and possibly word 3)

ACTION:

Causes an unconditional interpreting of the [true-option] field, based on the value of mm:

mm = 0 : [true-option] is an instruction table location; for this case, parsing resumes immediately at this new location.

mm = 1 : [true-option] is an error number; for this case, control passes from the parser to the error system, with this error number.

mm = 2 : [true-option] is the direct address in parser storage of an error number; control passes to the error system with the value of this error number.

mm = 3 : [true-option] is the indexed address in parser storage of an error number; control passes to the error system with the value of this error number.

4.2.8 General Parameter Table Instructions

4.2.8.1 General Instruction Form

TABLE FORM:

/ m m c c c c c p p p p p /	word 1
/ table parameter 1 /	1, 2, or 3 words
/ table parameter 2 /	1, 2, or 3 words
/ [true-option] /	1, or 2 words, only for conditional branch instructions.

ACTION:

This instruction form is used for all instructions that use 2 parameters as operands.

ppppp tells what the 2 parameter table forms look like:

aa = 7, ai = 8, a(ii) = 9, ac = 10,

ia = 11, ii = 12, i(ii) = 13, ic = 14,

(ii)a = 15, (ii)i = 16, (ii)(ii) = 17, (ii)c = 18.

where

a : direct address is parser storage (1 word)

i : indexed address in parser storage (2 words)

ii : doubly-indexed address in parser storage (3 words)

c : table constant packed in the parser table itself (1 word).

Based on the value of ppppp, the 2 parameters are decoded into both their addresses and their values; for this discussion let the 2 parameters be denoted as p1 and p2, and the decoding yields:

val(p1) and addr(p1) and

val(p2) and addr(p2).

These 2 parameters are then used according to the value of ccccc:

bceq = 0, bcne = 1, bcgt = 2, bcge = 3, bclt = 4, bcle = 5,

bcnotall = 6, bcnotany = t, bcall = 8, bcany = 8,

assign = 10, passign = 11, maskon = 12, maskoff = 13,

addit = 14, subit = 15.

The BC table instructions are described in section 4.2.8.2, and the environment-changing instructions are described in section 4.2.8.3.

4.2.8.2 Conditional Branch BC Table Instructions

ACTION:

Causes the comparison of the values of 2 parameters, X = val(p1), Y = val(p2).

This chart describes the TRUE condition requirements for each ccccc:

ccccc	condition	Condition TRUE if
0	bceq	$X = Y$
1	bcne	$X \neq Y$
2	bcgt	$X > Y$
3	bcge	$X \geq Y$
4	bclt	$X < Y$
5	bcle	$X \leq Y$
6	bcnotall	((X $mask$ Y) ≠ Y)
7	bcnotany	((X $mask$ Y) = 0)
8	bcall	((X $mask$ Y) = Y)
9	bcany	((X $mask$ Y) ≠ 0)

where "$mask$" performs the bit-wise "AND" of X and Y.

If the comparison of the 2 parameters yields FALSE, then the [true-option] is ignored, and parsing resumes with the next instruction in the table. If the comparison of the 2 parameters yields TRUE, the [true-option] is interpreted as follows:

mm = 0 : [true-option] is an instruction table location; for this case, parsing resumes immediately at this new location.

mm = 1 : [true-option] is an error number; for this case, control passes from the parser to the error system, with this error number.

mm = 2 : [true-option] is the direct address in parser storage of an error number; control passes to the error system, with the value of this error number.

mm = 3 : [true-option] is the indexed address on parser storage of an error number; control passes to the error system with the value of this error number.

4.2.8.3 Parser Environment-Changing Table Instructions

ACTION:

All of these instruction forms involve changing or modifying the value of the first parameter in the parser storage area. The following chart summarizes the actions of the different instructions:

ccccc	instruction	action
10	assign	val(pl) ←val(p2)
11	passign	val(pl) ←addr(p2)
12	maskon	val(pl) ←val(pl) \$union\$ val(p2)
13	maskoff	val(pl) ←val(pl) \$mask\$ (-val(p2))
14	addit	val(pl) ←val(pl) + val(p2)
15	subit	val(pl) ←val(pl) - val(p2)

where "\$mask\$" is the bit-wise "AND", "\$union\$" is the bit-wise "OR", and the (-val(p2)) for the maskoff instruction is the bit-wise complement of val (p2).

Before any of these instructions are executed by the parser, the old value of parameter 1 is traced if necessary (see section 3.3.9 for exceptions).

Note that the only current use of the PASSIGN table instruction is for passing parameters to semantic routines; there is no assembler source form of the PASSIGN instruction.

The following are some illegal table references:

st_idn (st_dvl(var)) ,where var is a <u>local</u> ALLOCATEd variable.

st_typ(1) ,constants **are** <u>Not</u> allowed in current version.

st_ub3 (udstp) ,there is no st_ub3!

CHAPTER 5.

5. THE COMPILER'S TABLE MAINTENANCE SYSTEM

5.1 Maintenance System's Purpose

Included as part of the compiler system that has been implemented on PLATO IV is a general compiler's table maintenance system. This maintenance system is designed to allow language implementors to completely specify all of the tables that are required for a particular language to be recognized and used as part of the computer science compiler system. The maintenance system also allows changes to be made and tested to existing "stable" versions of the compiler system tables without disturbing the "stable" version until the modifications have been throughly debugged.

The remainder of this chapter will discuss the utilization of this table maintenance system.

5.2 General Operation of the Maintenance System

The compiler's table maintenance system maintains a large dataset file that PLATO IV stores on a disk. Each "used" block on the dataset is associated with some particular language implementation for the compiler; all of the compiler tables for a given language are stored within the blocks that are allocated for that language.

A language implementor is allowed to modify any of the tables that are stored on the dataset for the language that is being implemented. Experimental versions of an existing language are easily created; these new versions are allocated their own disk space, which allows modifications to be made without disturbing the original version. Figure 5.1 shows all of the dataset FILE MANAGEMENT options that are available to a language implementor. Figure 5.2 shows an example of the dataset directory.

The actual compiler itself utilizes a set of tables that are stored in "common", which is an Extended Core storage file associated directly with the compiler lesson. The table maintenance system allows a language implementor to update the compiler's "common" tables from the copy of the tables that are stored in the dataset.

Thus, the maintenance system allows a language implementor to completely maintain the tables that are used by the compiler system.

5.3 Logging into the Maintenance System

The table maintenance system requires that each language in the dataset be code-word protected. When signing into the maintenance system, the proper code-word must be typed in before the tables for a language may be modified (figure 5.3 illustrates this process).

After the proper code-word has been entered, the language name must be specified (see figure 5.4).

Once the language name has been correctly typed in, the language implementor is officially logged into the table maintenance system. At this time, a number of options are available (see figure 5.5), including editing or assembling the syntax language source text. Access is also allowed to any of the FILE MANAGEMENT options mentioned above (figure 5.1).

Note that the table maintenance system also allows someone to sign into the system without requiring that a code-word be entered; this puts the person in an INSPECT ONLY mode, in which the person may examine any of the tables for a language, but is not allowed to make any change to any language.

F I L E   M A N A G E M E N T

a) Inspect directory
b) Edit a syntax file
c) Create a file
d) Extend a file
e) Shorten a file
f) Rename a file
g) Destroy a file
h) Change a file password
i) Create a copy of a file
j) Initialize dataset
k) Dataset unload--syntax source
l) Convert from 4 to 5 block common

Press -DATA- to change your password
Press -BACK- for TABLE BUILDER OPTIONS

FIGURE 5.1 FILE MANAGEMENT OPTIONS

D I R E C T O R Y

LANGUAGE	# TABLE BLOCKS	# SYNTAX BLOCKS	LAST EDITED BY
basic	5	8	milner(csa) on Ø5/27/75
fortran	4	9	milner(csa) on Ø5/27/75
pl1comp	5	9	tindall(csa) on Ø5/29/75
fmike	5	15	milner(csa) on Ø5/27/75
cobol	4	14	milner(csa) on Ø5/28/75
new.fort	5	9	tindall(csa) on Ø5/29/75
snobol	4	6	cupec(uicc) on Ø5/29/75
cobolnew	5	14	rush(rm) on Ø5/29/75
cblpics	4	1	milner(csa) on Ø5/27/75
lisp	5	4	milner(csa) on Ø5/27/75
pascal	5	7	schubert(cs491) on Ø5/29/75
pl1xcg	5	1Ø	milner(csa) on Ø5/27/75
lisp2	5	4	milner(csa) on Ø5/27/75
fortcg	4	9	nakamura(cs491) on Ø5/27/75
barnard	5	1	barnard() on Ø4/24/75
basiccopy	5	8	milner(csa) on Ø5/27/75
fort.comm	4	9	milner(csa) on Ø5/26/75

SPACE UTILIZATION (17 languages)
USED = 216 REMAINING = 34

FIGURE 5.2 An Example of the DATASET DIRECTORY

```
****************************************
***C.S. Compiler ****** Table Builder***
****************************************

Type in your File Security code, then press -NEXT-

     Press -NEXT- for INSPECT ONLY privledges.




              Press -BACK- to exit
```

FIGURE 5.3 Language CODE-WORD Entry

```
TABLE BUILDER OPTIONS:

Type in a language name:

      > pl1comp

Then press -NEXT-

Press -DATA- to change your password
Press -LAB- for file management options
```

FIGURE 5.4 Language NAME Entry

```
TABLE BUILDER OPTIONS:

Language   ***pl1comp***

   Last edited by *tindall(csa)* on * Ø5/29/75*.

Choose an option to proceed:

   1)   to edit syntax language source.
   2)   to assemble syntax language source.
   3)   to modify all other compiler tables.
   4)   to jump to the compiler.
   5)   for a lexical analysis diagram.

   Press -BACK- to choose another language
   Press -DATA- to change your password
   Press -LAB- for file management options
```

FIGURE 5.5 TABLE BUILDER OPTIONS

5.4 Preparing the Syntax Table for the Compiler System

Once a new language has been created on the dataset, the language implementor then proceeds with the "edit the syntax language source" table builder option; the initial version of the syntax source specification is then typed into the syntax blocks for the language.

When the syntax specification is complete, the table option "to assemble the syntax language source" should be attempted. This option assembles the source form into the compiler's table form; as the assembler is running, the current label in the syntax source specification is displayed to allow the language implementor to follow the progress of the assembler (see figure 5.6).

Any errors in the syntax specification will be flagged as they are detected; no attempt is made by the assembler to correct the error--the language implementor must return to the table builder, fix the indicated error, and attempt to reassemble the syntax source until no errors are detected by the assembler.

When a correctly assembled table has been prepared by the assembler, upon returning to the table maintenance system the language implementor should update the copy of the tables on the disk (see figure 5.7)

Once the syntax table is prepared, as well as all of the other compiler tables, the table builder option "to jump to the compiler" should be taken to initialize or update the compiler's actual "common" table version (note that this version of the compiler will also contain the appropriate set of TUTOR semantic routines for the given language (see section 3.3.8)). Then any logic errors in the syntax specification must be discovered and corrected by the language implementor.

ASSEMBLER
IS
RUNNING

The label currently being scanned is: arg1

FIGURE 5.6 Running The Assembler

```
Press -HELP1- to update your tables on the disk

Press -NEXT-  to forget about such ideas...
```

FIGURE 5.7 Updating the DISK Copy of the Tables

CHAPTER 6.

6. FUTURE DEVELOPMENT

The table-driven parser system that has been described in this paper works fairly well for the languages that have been implemented thus far (PL/I, FORTRAN, COBOL, BASIC). However, there are a few areas in which improvements could be made within both the actual table system, and the corresponding assembler language specifications.

One area that could be greatly improved is the handling of semantic routines in the system (section 3.3.8). When the table-driven system was first designed, very few semantic-type instructions were included (examples are ASSIGN, ADDIT, SUBIT, etc.) because it was not known exactly what instructions would be needed. Now that a number of languages have been implemented, the semantic routines that are used by these languages need to be surveyed very carefully so that the commonly-used routines can be included directly as new instructions in the system (examples might be instructions to open (close) a block, or to request that a dope vector be allocated and linked to a particular symbol table entry). Ultimately, the hope would be to eliminate nearly all actual uses of the SEMA instruction in the system, with the appropriate functions being accomplished more directly.

A second improvement to the system would be the development of a slightly higher-level form of the assembler language to be used in specifying the syntax/semantics of a programming language. For example, a simple looping-type construct would be very useful in the assembler.

Another useful addition would be to allow more complicated data-structures to be declared within the Parser's variable stack; for example, it would be useful to be able to easily create and manipulate linked-lists of ALLOCATEd variables on the parser's stack.

A final improvement, and by far the most difficult one, is to develop a program to convert a BNF-like representation of the syntactic/semantic requirements for a language into the table-form that is required by the compiler system. Although the most difficult improvement suggested here, this would also be the most useful because it would allow language designers to implement new languages without having a detailed knowledge of the actual parser table system that is used in the compiler.

LIST OF REFERENCES

[1] Conway, M.E., "Design of a Separable Transition-Diagram Compiler", CACM, Vol. 6, (July 1963), pp. 396-408.

[2] Eland, David, Forthcoming Ph.D. thesis on the GUIDE information-retrieval system, to be published in the summer of (1975).

[3] Lomet, D. B., "A Formalization of Transition-diagram Systems", Journal of the ACM, Vol. 20, No. 2, (April 1973), pp. 235-257.

[4] Nievergelt, J., Reingold, E. M., and Wilcox, T.R., "The Automation of Introductory Computer Science Courses," A. Gunther, et al. (editors), International Computing Symposium 1973, North-Holland Publishing Co., 1974.

[5] Sherwood, B. A., The TUTOR Language, Computer-based Education Research Laboratory and Department of Physics, University of Illinois, Urbana.

[6] Tindall, M. H., Forthcoming Ph.D. thesis on an interactive, compile-time table-driven error analysis system, to be published fall (1975).

APPENDIX

This appendix contains a sample assembler source language listing for a version of a subset of the PL/I programming language.

```
# CLASS VALUES
#DEFINE  OPUN=1
DEFINE   OPBIN=2,OPNOT=3,OPCOND=4,OPREL=5,OPRIF=6
DEFINE   STRING=7,ATTRIB=8
DEFINE   PUNCT=9,DCLVAR=10,CONST=11,RSWD=12,NODCL=13
DEFINE   LABEL=14,REF=15,ENTRYC=16,ARRAY=17
DEFINE   EXTERNAL=18,STRIF=19,STRCON=20
#
#ST+6IDN VALUES
# UDN FIELD+; ATTRIBUTE LISTS---AS FOLLOWS+;
# UDN=[ABCDEFGHIJKL]...
# L=FIXED;K=FLOAT;J=DECIMAL;I=CHARACTER;H=ENTRY;
DEFINE   +NUM= MASK(000000000111)  #NUMERIC TYPE IDENTIFIER
DEFINE   +CHAR=MASK(000000001000)  #CHAR. TYPE IDENIIFIER
#
# PDN VALUES
DEFINE   COLONX=105,PROCEDX=18,OPTIONSX=22,MAINX=23,SEMIX=106
DEFINE   EOX=83,GOX=7,GOTOX=2,IFX=0,TOX=14,THENX=10,ENDX=9
DEFINE   ELSEX=11,STOPX=6,DOX=1,BYX=15,WHILEX=13
DEFINE   LEFTP=100,RIGHTP=101,PUTX=5,GETX=4,SKIPX=12
DEFINE   LISTX=20,COMMAX=104,FLOATX=16,DECIMALX=17
DEFINE   DECLAREX=3
DEFINE   PAGEX=39,FIXEDX=40,CHARX=41,ENTRYX=42
DEFINE   RETURNSX=43,RETURNX=8,CALLX=19
DEFINE   CONCATX=70,SUBSTRX=36
DEFINE   VARYX=44
DEFINE   +PLUS=65,+MINUS=66 # FOR UNARY OPERS
# SEMA NUMBERS
DEFINE   ATTRTEST=1,ATTRSET=2,ATTRDFLT=3  ### ,MARKLAB=4
DEFINE   NEWLINE=5,POSLAB=6,INDENT=7,OUTDENT=8
DEFINE   SETBND=9,CHNGBND=10,GETDV=11 #,GETKID=12
DEFINE   SETLEN=13,LABCHK=14
#
#SYSTEM ERROR MSSG...
DEFINE   +SYSERR=999
#
# END OF MNEUMONIC CONSTANT DEFINITION
```

```
*
* GLOBAL VARIABLE ALLOCATION
ALLOCATE ERR,DV,VAR,DIM
* END ALLOCATIONS
*
* ERROR CLASS NAMES FOR AUTO ERROR SYSTEM
*CLASS NAME OPUN (+, -)
CLASS NAME OPRTN (ARITHMETIC OPERATOR)
CLASS NAME OPNOT (≥L≥)
CLASS NAME OPCOND (≥F≥ OR ≥G≥)
CLASS NAME OPREL (RELATIONAL OPERATOR)
CLASS NAME OPBIF (NUMERIC BUILT-IN FUNCTION)
CLASS NAME STRING(STRING)
CLASS NAME ATTRIB(ATTRIBUTE)
CLASS NAME PUNCT(PUNCTUATION SYMBOL)
CLASS NAME DCLVAR(DECLARED VARIABLE)
CLASS NAME CONST(NUMERIC CONSTANT)
CLASS NAME RSWD(RESERVED WORD)
CLASS NAME NODCL(UNDECLARED VARIABLE)
CLASS NAME LABEL(DEFINED LABEL)
CLASS NAME REF(LABEL REFERENCE)
CLASS NAME ENTRYC(ENTRY NAME)
CLASS NAME ARRAY(DECLARED ARRAY)
CLASS NAME EXTERNAL(EXTERNAL VARIABLE)
CLASS NAME STBIF(STRING BUILT-IN FUNCTION)
CLASS NAME STRCON (STRING LITERAL)
*END OF CLASS NAMES
*
*UDN MASK NAMES...
MASK NAME ↑CHAR (CHARACTER)
MASK NAME ↑NUM  (NUMERIC)
* END OF MASK NAMES
*
```

```
*
* INITIAL PROGRAM
BEGIN    BC NE,CLASS,NODCL,ERROR 75 *MUST HAVE ENTRY NAME
         ASSIGN CLASS,ENTRYC
         SCAN
         BC NE,PDN,COLONX,ERROR 3 *REQ ↑: HERE
         SEMA POSLAB
         SCAN
         BC NE,PDN,PROCEDX,ERROR 6 * REQ ↑P↑R↑O↑C HERE
         SCAN
         ASSIGN ERR,7 * REQ OPTIONS(MAIN) OR ; HERE
         BC NE,PDN,OPTIONSX,PROCEXT
         SCAN
         BC NE,PDN,LEFTP,ERROR 4 * REQ ( HERE
         SCAN
         BC NE,PDN,MAINX,ERROR 8 * REQ ↑M↑A↑I↑N HERE
         SCAN
         BC NE,PDN,RIGHTP,ERROR 5 * REQ ) HERE
         ASSIGN ERR,1  * REQ ; HERE
         SCAN
PROCSEMI BC NE,PDN,SEMIX,ERROR ERR
         SEMA NEWLINE
         SCAN
PROCLOOP BC EQ,PDN,ENDX,PROCEND * CHECK FOR ↑P↑R↑O↑C ↑E↑N↑D
         CALLI STMNT
         BC TRUE,PROCLOOP
PROCEND  SCAN
         BC NE,CLASS,ENTRYC,PROCSEM
         SCAN
PROCSEM  BC NE,PDN,SEMIX,ERROR 1 * REQ ; HERE
         SCAN
         FINAL PARSE STATE
         BC TRUE,ERROR 76 * ALL TEXT INSIDE PROC
PROCEXT  BC NE,PDN,LEFTP,PROCX2
PROCX1   SCAN
         BC NE,CLASS,NODCL,ERROR 25
*EVENTUALLY SEMANTIC ROUTINE HERE
         SCAN
```

```
        BC EQ,PDN,COMMAX,PROCX1
        BC NE,PDN,RIGHTP,ERROR 16
        SCAN
        ASSIGN ERR,31
*
PROCX2  BC NE,PDN,RETURNSX,PROCSEMI
        SCAN
        BC NE,PDN,LEFTP,ERROR 4
        SCAN
        BC NE,CLASS,ATTRIB,ERROR 24
        BC EQ,PDN,ENTRYX,ERROR 24
        SCAN
        BC NE,PDN,RIGHTP,ERROR 5
        SCAN
        BC TRUE,PROCSEMI
*
* END OF INITIAL PROGRAM
*
PROC    STMNT NAME(STATEMENT)
STMNT00 BC EQ,CLASS,RSWD,QUART2
        BC EQ,CLASS,DCLVAR,STMNT1
        BC EQ,CLASS,ARRAY,STMNT11
STMNT0  BC EQ,CLASS,NODCL,STLAB
        BC NE,CLASS,REF,STMNT2
* MUST BE A LABEL
STLAB   ASSIGN CLASS,LABEL
*  NO SEMA MARKLAB HERE...
        ASSIGN ST+60FF(UDSTP),ITPTP  *I.T.LABEL LOCATION.
        SCAN
        BC EQ,PDN,EQX,ERROR 88 * SEMAN. ERROR-MUST DCL VARS
        BC NE,PDN,COLONX,ERROR 3 * REQ :S HERE
        SEMA POSLAB
        SCAN
        BC TRUE,STMNT00
*
```

```
*ASSIGNMENT STMNT
STMNT11 ASSIGN DIM.ST+6DVL(UDSTP)
        BC MASK.ANY,UDN,+CHAR,STMNT1C
        BC MASK.NOTANY,UDN.+NUM,ERROR +SYSERR
        CALL SUBSCRLP
        BC TRUE.STMNT12
STMNT1  BC MASK.ANY,UDN,+CHAR,STMNT2C
        BC MASK,NOTANY,UDN,+NUM,ERROR +SYSERR
        SCAN
STMNT12 BC EQ,RDN,COLONX,ERROR 84 *SEMAN SRROR-DCLVAR NAME
        BC NE.RDN,EQX,ERROR 2 * REQ = HERE
        CALL EXPR
STMNT13 BC EQ,RDN,RIGHTP,ERROR 92 * EXPR HAS TOO MANY )
        BC NE.RDN,SEMIX,ERROR 1 * REQ ; HERE
        SEMA NEWLINE
        RET
*
STMNT1C CALL SUBSCRLP
        BC TRUE.STMNT3C
STMNT2C SCAN
STMNT3C BC EQ,RDN,COLONX,ERROR 84
        BC NE,RDN,EQX,ERROR 2  * REQ = HERE
        CALL STEXPR  *SCAN STRING EXPRESSION
        BC TRUE.STMNT13
*
STMNT2  BC EQ.CLASS.LABEL,ERROR 86 * DUPLICATE LABEL
        BC TRUE.ERROR 89  * BAD STMNT START
*
* LOOK AT FIRST WORD OF STMNT
*
QUART2  BC NE.RDN,DECLAREX,STGO
* +D+C+L STMNT
STDCL   ASSIGN ERR.RT+6KID(BLOCK) *NOT MNEMONIC
STDCL1  CALL PACKET
        BC EQ,RDN,COMMAX,STDCL1
        BC NE.RDN,SEMIX.ERROR 19 * ; OR ,
        SEMA ATTRDFLT(ERR)  *SET DEFAULT ATTRIBUTES...
        SEMA NEWLINE
        RET
```

```
*
STGO     BC NE,PDN,GOX,STGOTO
* ↑G↑O STMNT
         SCAN
         BC NE,PDN,TOX,ERROR 18 * REQ ↑T↑O HERE
         GOTO STGOTO1
*
STGOTO   BC NE,PDN,GOTOX,STIF
* ↑G↑O↑T↑O STMNT
         SCAN
STGOTO1 BC NE,CLASS,NODCL,STGOTO2
         ASSIGN CLASS,REF
STGOTO3 SCAN
         BC NE,PDN,SEMIX,ERROR 1 * REQ ; HERE
         SEMA NEWLINE
         RET
STGOTO2 BC EQ,CLASS,LABEL,STGOTO3
         BC EQ,CLASS,REF,STGOTO3
         BC EQ,CLASS,DCLVAR,ERROR 84 *INVALID LABEL-↑I↑D
         BC EQ,CLASS,ARRAY,ERROR 84
         BC EQ,CLASS,ENTRYC,ERROR 87 * CANT ↑G↑O↑T↑O ENTRY NAME
         BC TRUE,ERROR 17 * REQ A LABEL HERE
*
*
STIF     BC NE,PDN,IFX,STSTOP
* ↑I↑F STATEMENT
         CALL COND,THEN STIF2,STIFERR
STIFERR BC TRUE,ERROR 98 * ONLY EXPR IN COND
STIF2    BC EQ,PDN,RIGHTP,ERROR 97 * TOO MANY ) IN CONDEXPR
         BC NE,PDN,THENX,ERROR 9 * REQ ↑T↑H↑E↑N HERE
         CALL STMNT
         BC NE,PDN,ELSEX,STIF1
         CALL STMNT
STIF1    RETI
*
```

```
STSTOP   BC NE,PDN,STOPX,STDO
* ↑S↑T↑O↑P STMNT
         SCAN
         BC NE,PDN,SEMIX,ERROR 1 * REQ ; HERE
         SEMA NEWLINE
         RET
*
STDO     BC NE,PDN,DOX,STPUT
* ↑D↑O STMNT
         SCAN
         ASSIGN  ERR.12  *  REQ ↑IDENT,↑W↑H↑I↑L↑E OR ; HERE
         BC EQ,CLASS,DCLVAR,STDOSPEC
         BC NE,CLASS,ARRAY,STDOWHIL
         BC MASK,NOTANY,UDN,↑NUM,ERROR 34 *REQ NUM. VAR
         ASSIGN DIM,ST↑6DVL(UDSTP)
         CALL SUBSCRLP
* ITERATIVE LOOP WITH INDEX
STDOSPEC BC MASK,NOTANY,UDN,↑NUM,ERROR 34 *REQ NUM. VAR
         SCAN
         BC NE,PDN,EQX,ERROR 2 * REQ = HERE
         CALL EXPR
         BC EQ,PDN,RIGHTP,ERROR 92 * EXPR HAS TOO MANY )
         ASSIGN ERR. 13 * REQ ↑T↑O,↑B↑Y,↑W↑H↑I↑L↑E OR ; HER
STDOTO   BC NE,PDN,TOX,STDOBY
         CALL EXPR
         BC EQ,PDN,RIGHTP, ERROR 92 *EXPR HAS TOO MANY )
         ASSIGN ERR.14  * REQ ↑B↑Y,↑W↑H↑I↑L↑E OR ; HERE
STDOBY   BC NE,PDN,BYX,STDOWHIL
         CALL EXPR
         BC EQ,PDN,RIGHTP,ERROR 92 * EXPR HAS TOO MANY )
         ASSIGN ERR. 15  *  REQ ↑W↑H↑I↑L↑E OR ; HERE
STDOWHIL   BC NE,PDN,WHILEX,STDOSEMI
         SCAN
         BC NE,PDN,LEFTP,ERROR 4 * REQ ( HERE
         CALL COND,THEN STDOWH1,STDOWERR
STDOWERR  BC TRUE,ERROR 98 * ONLY EXPR IN COND
```

```
STDOWH1 BC NE,PDN, RIGHTP,ERROR 5 * REQ ) HERE
        SCAN
        BC EQ,PDN,RIGHTP,ERROR 97 *TOO MANY ) IN CONDEXPR
        ASSIGN ERR, 1  * REQ ; HERE
STDOSEMI  BC NE,PDN,SEMIX,ERROR ERR
        SEMA INDENT   *  INDENT AND NEWLINE
        SCAN
* LOOP UNTIL ↑E↑N↑D STMNT
STDOLOOP  BC EQ,PDN,ENDX,STDOEND
        CALLI STMNT
        BC TRUE,STDOLOOP
STDOEND SCAN
        BC NE,PDN,SEMIX,ERROR 1 * REQ ; HERE
        SEMA OUTDENT
        RET
*
STPUT   BC NE,PDN,PUTX,STGET
* ↑P↑U↑T STMNT
        ASSIGN ERR,10 * REQ ↑S↑K↑I↑P,↑P↑A↑G↑E OR ↑L↑I↑S↑T HERE
        SCAN
        BC NE,PDN,SKIPX,STPUTPGE
* ↑S↑K↑I↑P PARAMETERS
        SCAN
        ASSIGN ERR,20 * REQ ( ,;, OR ↑L↑I↑S↑T HERE
        BC NE,PDN,LEFTP,STPUTOPT
        CALL EXPR
        BC NE,PDN,RIGHTP,ERROR 5 * REQ ) HERE
        SCAN
        BC EQ,PDN,RIGHTP,ERROR 92 * EXPR HAS TOO MANY )
        BC TRUE,STPUTNOP
STPUTPGE BC NE,PDN,PAGEX,STPUTLST
        SCAN
STPUTNOP ASSIGN ERR,11 * REQ ↑L↑I↑S↑T OR ; HERE
STPUTOPT BC EQ,PDN,SEMIX,STPUTOUT
STPUTLST BC NE,PDN,LISTX,ERROR ERR
        SCAN
        BC NE,PDN,LEFTP,ERROR 4 * REQ ( HERE
*
```

```
STPUTLP SCAN
######HOW ABOUT CHAR. ARRAY↑/↑/↑/
        BC NE.CLASS.ARRAY.STPUTEX
        ASSIGN DIM.ST↑6DVL(UDSTP)
        SCAN
        BC NE.PDN.LEFTP.STPUTDL
        CALL SUBSCRIPT
*STPUTON BC EQ.CLASS.OPUN.STPUTXS
STPUTON BC NE,CLASS.OPBIN.STPUTDL
STPUTXS SCAN
STPUTEX CALLI OUTEXPR,THEN STPUTDL,STPUTDL
STPUTDL BC EQ.PDN.COMMAX.STPUTLP
        BC NE.PDN.RIGHTP,ERROR 16 * REQ , OR ) HERE
        SCAN
STPUTOUT BC NE.PDN.SEMIX.ERROR 1 * REQ ; HERE
        SEMA NEWLINE
        RET
STGET   BC NE.PDN.GETX.STCALL
* ↑G↑E↑T STMNT
        SCAN
        BC NE.PDN,LISTX.ERROR 11
        SCAN
        BC NE.PDN.LEFTP.ERROR 4
STGETLP SCAN
        BC EQ.CLASS.DCLVAR.STGET9
        BC NE.CLASS,ARRAY.ERROR 29
        ASSIGN DIM.ST↑6DVL(UDSTP)
        SCAN
        BC NE.PDN.LEFTP.STGET10
        CALL SUBSCRIPT
        BC TRUE.STGET10
STGET9  SCAN
STGET10 BC EQ.PDN.COMMAX.STGETLP
        BC NE.PDN.RIGHTP,ERROR 16
        SCAN
        BC NE.PDN.SEMIX.ERROR 1 * REQ ; HERE
        SEMA NEWLINE
        RET
```

```
STCALL   BC NE,PDN,CALLX,STENTRY
*↑C↑A↑L↑L STATEMENT
         SCAN
         BC NE,CLASS,NODCL,CALL0
         ASSIGN CLASS,EXTERNAL
         BC TRUE,CALL01
CALL0    BC EQ,CLASS,LABEL,ERROR 86
         BC EQ,CLASS,DCLVAR,ERROR 85
         BC NE,CLASS,EXTERNAL,ERROR 75
CALL01   SCAN
         BC EQ,PDN,SEMIX,CALL2
         BC NE,PDN,LEFTP,ERROR 32
CALL1    CALL EXPR
         BC EQ,PDN,COMMAX,CALL1
         BC NE,PDN,RIGHTP,ERROR 15
         SCAN
         BC NE,PDN,SEMIX,ERROR 1    *MISSING ;
CALL2    SEMA NEWLINE
         RET
STENTRY  BC NE,PDN,ENTRYX,STRET
*↑E↑N↑T↑R↑Y STATEMENT
         SCAN
         BC EQ,PDN,SEMIX,ENTRY2
         BC NE,PDN,LEFTP,ERROR 32   * ( OR ;
ENTRY1   SCAN
         BC NE,CLASS,ATTRIB,ERROR 24
         SCAN
         BC EQ,PDN,COMMAX,ENTRY1
         BC NE,PDN,RIGHTP,ERROR 16
         SCAN
         BC NE,PDN,SEMIX,ERROR 1
ENTRY2   SEMA NEWLINE
         RET
```

```
STRET    BC NE,PDN,RETURNX,ERROR 29   *BAD STATEMENT START
*+R+E+T+U+R+N STATEMENT
         SCAN
         BC EQ,PDN,SEMIX,RETRN1
         BC NE,PDN,LEFTP,ERROR 32
         CALL EXPR
         BC NE,PDN,RIGHTP,ERROR 5   * )
         SCAN
RETRN1   SEMA NEWLINE
         RET
END      PROC               *STMNT
*
PROC     PACKET NAME(IDENTIFIER DECLARATION)
         ALLOCATE LIMIT,ATTR
         ASSIGN ATTR,0
         ASSIGN LIMIT,BT+6KID(BLOCK)
         BC EQ,PDN,LEFTP,DCLLST
         BC NE,CLASS,NODCL,ERROR 26
         ASSIGN VAR,UDSTP
         SCAN
         BC NE,PDN,LEFTP,SCLR
         ASSIGN ST+6TYP(VAR),ARRAY
         SEMA GETDV(VAR,DV)
         ASSIGN DIM,1
         CALL BOUND
         BC NE,PDN,COMMAX,DIM1
         ASSIGN DIM,2
         CALL BOUND
DIM1     ASSIGN ST+6DIM(DV),DIM
         BC NE,PDN,RIGHTP,ERROR 5
         SCAN
         BC TRUE,PACKET1
SCLR     ASSIGN ST+6TYP(VAR),DCLVAR
         BC TRUE,PACKET1
DCLLST   CALL PACKET
         BC EQ,PDN,COMMAX,DCLLST
         BC NE,PDN,RIGHTP,ERROR 5
         SCAN
```

```
PACKET1 BC NE,CLASS,ATTRIB,PACKET2
* STOFF IS ATTRIB. MASK BIT, STLEN IS ATTRIB. CONFLICT BITS
PK1     SEMA ATTRTEST(LIMIT,ATTR,ST↑6LEN(PDSTP),ST↑6OFF(PDSTP))
        BC NE,TEMP1,0,ERROR TEMP1 *ATTRIB. CONFLICT
        BC NE,PDN,CHARX,PACKET11
        SCAN
        BC NE,PDN,LEFTP,ERROR 4
        SCAN
        BC NE,CLASS,CONST,ERROR 33
        SEMA SETLEN(LIMIT)
        SCAN
        BC NE,PDN,RIGHTP,ERROR 5
PACKET11 SCAN
        BC TRUE,PACKET1
PACKET2 SEMA ATTRSET(LIMIT,ATTR)
        RETI
END     PROC
*
PROC    BOUND NAME(ARRAY BOUND)
        SEMA SETBND(DV,DIM)
        CALLI EXPR
        BC NE,PDN,COLONX,BND9
        SEMA CHNGBND(DV,DIM)
        SEMA SETBND(DV,DIM)
        CALL EXPR
BND9    RETI
END     PROC
*
PROC    SUBSCRLP NAME(SUBSCRIPT LIST)
        ALLOCATE NUM
        BC NE,PDN,LEFTP,ERROR 27
        SCAN
*
ENTRY   SUBSCRIPT NAME(SUBSCRIPTS)
        ASSIGN NUM,ST↑6DIM(DIM)
        CALLI EXPR
        BC EQ,NUM,1,SUB1
        BC NE,PDN,COMMAX,ERROR 28
```

```
         CALL EXPR
SUB1     BC NE.PDN.RIGHTP.ERROR 5
         RET
END      PROC
*
PROC     EXPR NAME(NUMERIC EXPRESSION)
         CALLI OPND
         BC TRUE.OPERX
OPER1    CALL OPND
********
ENTRY    OPER NAME(NUMERIC EXPRESSION)
*OPERX    BC EQ.CLASS.OPUN.OPER1
OPERX    BC EQ.PDN.CONCATX.ERROR 35 * NEEDS NUMER. OPERATOR
         BC EQ.CLASS.OPBIN.OPER1
         RETI
END      PROC              *EXPR
*
PROC     OPND NAME(NUMERIC OPERAND)
OPND1    BC EQ.CLASS.DCLVAR.OP1
         BC EQ.CLASS.CONST.OP0
         BC NE.CLASS.ARRAY.OP2
         BC MASK.NOTANY.UDN.↑NUM.ERROR 36 *NUMERIC OPERAND
         ASSIGN DIM.ST↑6DVL(UDSTP)
         CALL SUBSCRLP
         RETI
OP1      BC MASK.NOTANY.UDN.↑NUM.ERROR 36
OP0      RET
OP2      BC NE.CLASS.OPBIN.OP3
         BC EQ.PDN.↑PLUS.OP21
         BC NE.PDN.↑MINUS.ERROR 36
OP21     GOTO OPND1
OP3      BC EQ.PDN.LEFTP.OP5
         BC NE.CLASS.OPRIF.OP4
         ASSIGN DIM.ST↑6OFF(PDSTP)
         CALL ARGLIST
         RETI
```

```
OP4      BC NE.CLASS,STBIF,ERROR 36
         ASSIGN DIM,ST+6OFF(PDSTP)
         BC EQ.PDN,SUBSTRX,ERROR 36
         CALL STARGLST
         RETI
OP5      CALL EXPR
         BC NE.PDN,RIGHTP,ERROR 5
OP6      RET
END      PROC             *OPND
*
PROC     ARGLIST NAME(ARGUMENT LIST)
         ALLOCATE NUM
         ASSIGN NUM,DIM
         BC NE.PDN,LEFTP,ERROR 30
         CALL EXPR
         BC EQ.NUM,1,SUB11
         BC NE,PDN,COMMAX,ERROR 28
         CALL EXPR
SUB11    BC NE.PDN,RIGHTP,ERROR 5
         RET
END      PROC
*
PROC     COND NAME(CONDITIONAL EXPRESSION)
CON2     BC EQ.PDN,LEFTP,PAR
         CALLI EXPR
CONDREL BC EQ.CLASS,OPREL,EXP2
         RETI  2          *EXPRESSION
* PICK UP EXPR FOLLOWING RELOP
EXP2     CALL EXPR
COND+/    BC NE.CLASS,OPCOND,CONDNO
         GOTO CON2
CONDNO   RETI   1         *CONDITION
*
* BEGINS WITH (. +LOOK FOR COND OR EXPR
PAR      CALL  COND,THEN PCOND,PEXPR
*
* PARENTHESIZED COND FOUND
PCOND    BC NE.PDN,RIGHTP,ERROR  96 * COND HAS TOO MANY (
         GOTO COND+/
```

```
* PARENTHESIZED EXPR FOUND
PEXPR    BC NE,PDN,RIGHTP,ERROR  91 * EXPR HAS TOO MANY (
         CALL OPER
         BC TRUE,CONDREL
END      PROC             *COND
*
PROC     STEXPR NAME(CHARACTER EXPRESSION)
         CALLI STOPND
         BC TRUE,STOPER
STOPER1  CALL STOPND
STOPER   BC EQ,PDN,CONCATX,STOPER1
         RETI
END      PROC
*
PROC     STOPND NAME(CHARACTER OPERAND)
         BC EQ,CLASS,DCLVAR,STOP1
         BC EQ,CLASS,STRCON,STOP0
         BC NE,CLASS,ARRAY,STOP2
         BC MASK,NOTANY,UDN,↑CHAR,ERROR 37 *REQ CHAR OPERAND
         ASSIGN DIM,ST↑6DVL(UDSTP)
         CALL SUBSCRLP
         RETI
STOP1    BC MASK,NOTANY,UDN,↑CHAR,ERROR 37
STOP0    RET
STOP2    BC NE,PDN,SUBSTRX,STOP3
         SCAN
         BC NE,PDN,LEFTP,ERROR 30
         CALL STEXPR
         BC NE,PDN,COMMAX,ERROR 28
         CALL EXPR
         BC EQ,PDN,RIGHTP,STOP0
         BC NE,PDN,COMMAX,ERROR 28
         CALL EXPR
         BC NE,PDN,RIGHTP,ERROR 5
         RET
STOP3    BC NE,PDN,LEFTP,ERROR 37
         CALL STEXPR
         BC NE,PDN,RIGHTP,ERROR 5
         RET
END      PROC *STOPND
```

```
#
PROC     STARGLST NAME(ARGUMENT LIST)
         ALLOCATE NUM
         ASSIGN NUM,DIM
         BC NE,PDN,LEFTP,ERROR 30
         CALL STEXPR
         BC EQ,NUM,1,STSUB11
         BC NE,PDN,COMMAX,ERROR 28
         CALL STEXPR
STSUB11  BC NE,PDN,RIGHTP,ERROR 5
         RET
END      PROC #STARGLST
#
#THESE PROCS DUPLICATE A ↑L↑O↑T OF CODE FOUND ELSEWHERE.
# ↑THEY WILL BE GREATLY SHORTENED AS SOON AS ↑E↑N↑T↑R↑Y IS
#IMPLEMEMTED.
PROC     PUTEXPR NAME(EXPRESSION) RETURN 2
         CALLI ANYOPND,THEN PEXPRST,PEXPRNU
PEXPRST1 CALL STOPND
PEXPRST  BC EQ,PDN,CONCATX,PEXPRST1
         RETI 1
PEXPRNU1 CALL OPND
#PEXPRNU BC EQ,CLASS,OPUN,PEXPRNU1
PEXPRNU  BC EQ,PDN,CONCATX,ERROR 35  #NEEDS NUMERIC OPER.
         BC EQ,CLASS,OPBIN,PEXPRNU1
         RETI 2
END      PROC   #NUMERIC ↑O↑R STRING EXPRESSION
#
PROC     ANYOPND NAME(OPERAND) RETURN 2
AOPND1   BC EQ,CLASS,DCLVAR,AOP1
         BC EQ,CLASS,CONST,AOPNU
         BC EQ,CLASS,STRCON,AOPST
         BC NE,CLASS,ARRAY,AOP2
         ASSIGN DIM,ST↑6DVL(UDSTP)
         BC MASK,ANY,UDN,↑CHAR,AOPST1
         BC MASK,NOTANY,UDN,↑NUM,ERROR ↑SYSERR
         CALL SUBSCRLP
         RETI 2
```

```
AOPST1   CALL SUBSCRLP
         RETI 1
AOP1     BC MASK,ANY,UDN,↑CHAR,AOPST
         BC MASK,NOTANY,UDN,↑NUM,ERROR ↑SYSERR
AOPNU    RET 2
AOPST    RET 1
AOP2     BC NE,CLASS,OPBIN,AOP3
         BC EQ,PDN,↑PLUS,AOP21
         BC NE,PDN,↑MINUS,ERROR 25
AOP21    CALL OPND
         RETI 2
AOP3     BC EQ,PDN,LEFTP,AOP5
         ASSIGN DIM,ST↑6OFF(PDSTP)
         BC NE,CLASS,OPBIF,AOP4
         CALL ARGLIST
         RETI 2
AOP4     BC NE,CLASS,STBIF,ERROR 25
         BC EQ,PDN,SUBSTRX,AOPST2
         CALL STARGLST
         RETI 2
AOPST2   SCAN
         BC NE,PDN,LEFTP,ERROR 30
         CALL STEXPR
         BC NE,PDN,COMMAX,ERROR 28
         CALL EXPR
         BC EQ,PDN,RIGHTP,AOPST
         BC NE,PDN,COMMAX,ERROR 28
         CALL EXPR
         BC NE,PDN,RIGHTP,ERROR 5
         RET 1
AOP5     CALL PUTEXPR,THEN AOP6,AOP7
AOP6     BC NE,PDN,RIGHTP,ERROR 5
         RET 1
AOP7     BC NE,PDN,RIGHTP,ERROR 5
         RET 2
END      PROC
*
END      SYNA
```

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-75-745	2.	3. Recipient's Accession No.

4. Title and Subtitle

An Interactive Table-Driven Parser System

5. Report Date: August 1975

6.

7. Author(s)

Michael Harry Tindall

8. Performing Organization Rept. No. UIUCDCS-R-75-745

9. Performing Organization Name and Address

University of Illinois at Urbana-Champaign
Department of Computer Science
Urbana, Illinois 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.

NSF Grant EC-41511

12. Sponsoring Organization Name and Address

IBM Corporation
Thomas J. Watson Research Laboratory
Yorktown Heights, New York 10598

13. Type of Report & Period Covered

Master of Science Thes

14.

15. Supplementary Notes

National Science Foundation, Wasington, D.C. 20550

16. Abstracts

The design and implementation of the parser system for a multi-lingual compiler on the PLATO IV computer-aided instructional system is described. A brief discussion of the formal transition diagram parsing method is followed by the description of the assembler language used to specify the syntax of a programming language and the description of the actual parser tables.

17. Key Words and Document Analysis. 17a. Descriptors

Compilers, Parser Systems, Table Driven, Interactive Compiling, PLATO IV

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Release Unlimited	UNCLASSIFIED	90
	20. Security Class (This Page) UNCLASSIFIED	22. Price --------